

THE DESIGN OF AN OUTPUT DATA COLLECTION FRAMEWORK FOR NS-3

L. Felipe Perrone

Dept. of Computer Science
Bucknell University
Lewisburg, PA 17837, USA

Thomas R. Henderson
Mitchell J. Watrous

Dept. of Electrical Engineering
University of Washington
Seattle, WA, 98195, USA

Vinícius Daly Felizardo

Institute of Computing
Universidade Estadual de Campinas
Campinas, SP, BRAZIL

ABSTRACT

An important design decision in the construction of a simulator is how to enable users to access the data generated in each run of a simulation experiment. As the simulator executes, the samples of performance metrics that are generated beg to be exposed either in their raw state or after having undergone mathematical processing. Also of concern is the particular format this data assumes when externalized to mass storage, since it determines the ease of processing by other applications or interpretation by the user. In this paper, we present a framework for the *ns-3* network simulator for capturing data from inside an experiment, subjecting it to mathematical transformations, and ultimately marshaling it into various output formats. The application of this functionality is illustrated and analyzed via a study of common use cases. Although the implementation of our approach is specific to *ns-3*, this design presents lessons transferrable to other platforms.

1 INTRODUCTION

In computer simulation, data is both cause and consequence. The abstractions of a system or of a process that are used in simulation, known as *models*, are derived from the analysis of significant body of data. This “raw material” is most often acquired through heavy, challenging work and serves as the input that drives modeling activities. Once models are defined and subjected to an iterative process of validation and verification, their computational implementations are used in experiments executed by a simulator, which produces output data. This second body of data is processed and analyzed to yield answers for questions posed at the start of the simulation study. The collection and the processing of *input* and *output* data present significant challenges, each in its own way. In this paper, we focus exclusively on problems related to the generation and, to some extent, on the processing of output data.

Many simulators are constructed specifically for one particular simulation study and their construction tends to pay little regard to software engineering practices that promote flexibility and extensibility. The mechanisms for generation of output data, in these simulators, can be as simple as the direct use of a programming language’s I/O functionality to generate streams of samples of a few metrics of interest in the model. The instrumentation of code to externalize these samples may be done on a case by case basis, without the application of design patterns of any kind. Moreover, much of the processing of these samples

may be done internally to the simulator and often with code written “from scratch.” It comes as no surprise that many studies derived from simulators like these lack credibility, as they are prime examples of the situations identified as pitfalls by Uhrmacher (2012).

The situation is quite different in the case of simulators whose lifespan outlasts that of a single study, where one may find sophisticated, reliable functionality for data collection and processing. Still, as indicated by Ribault et al. (2010), many simulators provide APIs for data collection that require the mixing of instrumentation code into model code. There are two dangers in this approach. On the one hand, the model programmer may “overinstrument” the code to provide proactively for all possible and imaginable observation needs. This practice could lead to longer simulation runs that generate much more data than what is required for the average study. On the other hand, “underinstrumenting” models could require their future users to dive in and modify code, what could possibly introduce errors in the model and compromise experiments. In any case, the fact is that the quality of a simulation study is determined not only by how adequate the granularity and the amount of data collected are, but also by how easy it is to reproduce it. Kurkowski, Camp, and Colagrosso (2005) call attention to the importance of bundling data collection and output analysis into the simulation run, which can benefit reproducibility.

Recent literature, such as (Helms et al. 2012), indicate a growing attention to the search for simulator designs that include data collection mechanisms which are easy to use, flexible, and efficient, both in terms of memory and run time, and which can be coupled with some type of output analysis. In this paper, we present a framework for the *ns-3* network simulator that works toward these requirements. Although our framework relies on specific *ns-3* functionality, we expect that its underlying design provides lessons that can be generalized for application in other simulation platforms.

The remainder of the paper is structured as follows. Section 2 establishes a context for our contribution by presenting a few notable examples of related work. Section 3 expands this background by exposing the reader to the fundamentals of the design of *ns-3* and to the motivation behind our work. Section 4 introduces design goals that drove the design of this data collection framework and also the foundation classes that define it. Section 5 discusses implementation issues and shows examples of the application of the framework. Finally, Section 6 outlines ongoing work and future directions to conclude the paper.

2 RELATED WORK

Different simulators have different mechanisms for generating, collecting, and processing output data. In the field of network simulation, there have been several important developments toward augmenting or enhancing the functionalities provided natively by specific simulators. In this section, we discuss a few notable examples.

The *ns-2* network simulator provides only two mechanisms for data collection: *traces*, which relate to the occurrence of packet events (e.g., transmission, dropping) and *monitors*, which relate to statistics related to packet events (e.g., counters, rates). Both of these record data to files following a well-specified syntax. Cicconetti, Mingozzi, and Stea (2006) identified three major issues with this approach: logs record only events related to packet transmissions, getting to the data of interest is not always easy, and the file write operations add considerable run time overhead. The authors addressed these issues in the construction of the *ns2measure* framework. It offers a standard mechanism that allows the logging of general events, run time efficient data collection, and high quality infrastructure for output data analysis and report generation. The data collection portion of this framework defines *probes* (function calls inserted in the C++ code of *ns-2* models), which deposit samples of values into their `Stat` class for statistical processing. The static function `Stat::put()` is used to instrument the code of the model for data collection. The body of data accumulated during the simulation is analyzed statistically by another component of the framework, which also controls the execution of multiple, independent replications of the experiment. The concept of probe is noteworthy: it defines an instrument to extract data of various types at arbitrary points in time during the execution of a simulation. The notion of making a direct connection between a probe and mechanism

for statistical processing during the simulation run is also interesting. The downside of this approach is the need for users to mix code for data collection into *ns-2* model code.

The method proposed by Ribault et al. (2010) uses the concept of *separation of concerns* to avoid the intertwining of data collection code and simulation model code. Their *Open Simulation Instrumentation Framework (OSIF)* introduces interesting ideas. The separation of instrumentation code from model code enhances the readability of models and allows for models to be reused in different simulation studies. This practice has the consequence of avoiding the “overinstrumentation” of model code, often caused by overzealous developers and which results in the collection of much more data than what is of interest to the experiment. OSIF provides a general-purpose solution that uses *AspectJ*, an extension of Java for *Aspect Oriented Programming*, to separate model from data collection instruments. This solution is based on the concept of *collector*, which is analogous to what *ns2measure* calls *probe*: software entities that extract raw data from the simulation.

If “output analysis is the downfall of many simulation studies”, as Kurkowski, Camp, and Colagrosso (2005) state, it is important to make the task as painless as possible and perhaps to embed it in the simulation run. This is achieved in OSIF’s architecture, where collectors pass data to *processors*, which filter and aggregate the input they receive. A processor is a node in a directed graph; it receives input data, performs some computation, and emits output that may go into another node for further action. Processors can be arranged into hierarchies to accomplish complex computations built as a sequence of simple functions that are encapsulated into nodes. (An interesting property of this type of live analysis of data is that it may be implemented by multiple threads, to take advantage of parallelism available in the computing platform.) The online filtering and aggregation of simulation data can reduce the run time performance hit of I/O and also the storage requirements.

A similar approach in collection, aggregation and filtering is proposed by Helms et al. (2012). The authors define a language for data collection and processing for the JAMES II framework, which also embodies the principle of separation of concerns in an architecture made flexible by the use of plug-in components. This language extends JAMES II’s functionalities for data collection and online processing, which is decoupled both from the model and the experiment. Differently from OSIF, which relies on an aspect oriented language, JAMES II uses plug-ins to implement data collection and processing. The language used in this method draws inspiration from SQL and provides structures for selecting what to observe (*INSTRUMENT*) and for describing how to process collected data (*OBSERVE*). With a simple and yet expressive syntax, the language allows for statistical aggregation functions to be applied over attribute values in the model, in a style that resembles functional reactive programming, and allows for the definition of the frequency of observations. Although this approach seems promising, until now, it has only been implemented to support models in systems biology.

This paper’s contribution has points in common with *ns2measure*, OSIF, and JAMES II, as indicated in subsequent sections. The distinguishing points in the framework we present are multiple. Our primary goal is to support the *ns-3* simulator by extending its original mechanism for data collection with added control and functionality for the online processing of samples. The framework is implemented by a small set of foundational classes that can be extended to specialize the functionalities offered. These classes must be instantiated in the source code of the simulation script that defines the experiment and the framework expects that models contain minimal instrumentation to generate the output data of interest. Although our design doesn’t apply the separation of concerns principle, this framework places the code for data collection and processing not inside models, but in the *ns-3* script that defines the experiment. Finally, the design of our classes follows the *observer* design pattern (Gamma et al. 1995) and therefore implements the reactive programming paradigm. We elaborate on these points in Section 4, after introducing the context and the motivation for the work.

3 NS-3 AND SAFE

The design goals of *ns-3* have included providing the simulator core “with tools to that allow for highly customizable extraction of event logs and output statistics, to provide a framework for managing large numbers of simulation runs and output data, and to allow third-party analysis tools to be used where possible” (Riley and Henderson 2010). To this end, the project provides different kinds of facilities for data collection. It allows users to create ASCII trace files (similar to those of its predecessor *ns-2*) and packet capture trace files in the `pcap` format, which can be analyzed by external tools such as Wireshark and `tcpdump`. Additionally, data collection and processing is supported by two other types of facilities:

- *Trace sources* are the mechanism with which models can be instrumented to generate data at the occurrence of an event. The authors of models can use trace sources as producers that expose data of interest to another function. The facilities `TracedValue` and `TracedCallback` allow changes to the value of a class or primitive type, or selected events, to trigger the invocation of a registered function which receives a data value for processing.
- *Trace sinks* are functions that act as the consumers to the events and the data generated by trace sources.

This separation between producers and consumers of events and data decouples the modules that export trace sources from those that define trace sinks. This decoupling relates to the principle of separation concerns. In practice, it enables the separate compilation of modules that export trace sources (models) from the user script (*ns-3* 2013b). The model authors can instrument their code with various trace sources and each of them is used only when a user’s script connects one or more trace sinks to them. There is no escaping the fact, however, that when a model is not instrumented to generate the data of interest via a trace source, the user needs to plumb into model code to add that instrumentation.

Even though trace sources and trace sinks go a long way to provide the infrastructure for data collection and processing, they aren’t exactly a panacea. The user simulation script must still construct custom code for trace sinks, which can often be complex, hard to write, and repetitive. Once connected, trace sources and trace sinks are active for the duration of the simulation run: it is not possible to enable and disable them dynamically. Additionally, there is no provision to determine programmatically the simulation time at which observations of a value are generated. The basic `TracedValue` and `TracedCallback` infrastructure is purely event-driven. We realized also that leaving the creation of trace sinks entirely to users leads many to have to reinvent the proverbial wheel on their own, rather than promote the reuse of well designed, well tested code for processing samples of data and for marshaling them into various types of output format. The motivation for the creation of our *Data Collection Framework* (DCF) for *ns-3* has been to augment the functionality of the core of the simulator so as to address these issues.

The development of the DCF has occurred within the broader context of another project that supports *ns-3*. The *Simulation Automation Framework for Experiments* (SAFE) is a collection of tools external to the simulator that provide assistance to novice and to experienced users in following best practices in the simulation workflow. We refer the interested reader to Perrone, Main, and Ward (2012) for details on the design and the implementation of SAFE. We note that while the DCF attends to the needs of those who use *ns-3* by itself, it also supports the integration of the simulator with SAFE, which receives data generated and pre-processed by the simulation run to marshal into a database for persistent storage. The implementation of accurate methods for output data processing provided as infrastructure that is easy to use meets one of the main objectives of the SAFE project, which is to enhance the credibility of simulation experiments by addressing some of the issues identified by Kurkowski, Camp, and Colagrosso (2005).

4 DESIGN OF THE FRAMEWORK

The DCF comprises classes which cooperate toward creating a directed graph that reacts to changes in data collected from the simulation run and eventually produces data for output. The observations generated

in the *ns-3* script propagate through this graph, where each node performs some type of filtering and/or aggregation function. At the “end” of the graph, processed observations are marshaled into output formats chosen by the user.

In its essence, the DCF takes inspiration from dataflow concepts and relates to the ideas from *reactive programming*, which follows the *Observer* object-oriented design pattern (Gamma et al. 1995). The Observer pattern defines two roles for objects: **subject** and **observer**. The subject can be seen as the producer of data, which is consumed by one or more observers. A change of state in a subject causes a notification to be sent to all its subscribed observers, which query the subject for its state and use that result to update their own states. The variation on this communication used in the design of the DCF is the **push model**, in which subjects pass down to observers their state embedded in the notification. In the remainder of this section we discuss the four base classes that constitute the framework and provide a simple example to illustrate their use.

4.1 DcfObject

The fundamental class in the DCF is `DcfObject`. All other DCF classes are derived from this, which implements functionality to allow the dynamically enabling and disabling of the generation of output data in each individual object.

4.2 Probe

The DCF class which instruments *ns-3* models for data collection is `Probe`; in this sense, it implements the concept with the same name in *ns2measure* (Cicconetti, Mingozi, and Stea 2006). `Probe` can be seen as a wrapper for the standard *ns-3* trace source, which being derived from `DcfObject` can be enabled or disabled at any time in the simulation. In other words, it can be seen as a “controllable trace source.” Whenever the trace source encapsulated in the `Probe` generates a new value, this propagates down to any number of objects registered with it as observer. In the context of the Observer pattern, a `Probe` always plays the role of subject, however, there is no separation of notification and data transmission to observers connected downstream. The transfer of data from `Probe` to its observers happens by the standard mechanics of *ns-3* trace sources, that is, by callback functions registered at the time the connection is defined. Alternatively, in select cases, `Probe` can be configured to poll data rather than asynchronously wait for trace sources to fire.

`Probe` can be subclassed to deal with data types of varying complexity. At the same time that we can have probes that emit values of primitive data types such as `double`, we can also have probes that handle packets. Probes which deal with non-primitive data types such as this can be specialized to extract one or more of its instance fields. To each one of the values generated by a probe there corresponds one trace source. For instance, our current implementation offers a `PacketProbe` class which exports two trace sources: one with the same packet it received from upstream and another with the byte count in the packet, represented as an unsigned integer.

ns-3 implements a configuration namespace that identifies trace sources by “context strings,” which are hierarchical constructs analogous to paths in a file system tree. For instance:

```
/NodeList/0/$ns3::Ipv4L3Protocol/Tx
```

refers to a trace source object named `Tx` contained in the instance of model `Ipv4L3Protocol` for a network node identified by `0`, within a container of nodes `NodeList`. We use this namespace functionality to allow user scripts to connect probe objects easily to arbitrary trace sources exported in models.

Recent development on probes includes the following derivations from the `Probe` base class:

- `UInteger8Probe`, `UInteger16Probe`, and `UInteger32Probe` are designed to interoperate with trace sources that export data types `uint8_t`, `uint16_t`, and `uint32_t`, respectively. The

trace source exported by these probe classes emit a value when either the input trace source emits a new value, or a call to the `Probe` class' `SetValue (v)` is invoked to push out a given value v .

- `PacketProbe` exports two trace sources. The first is a trace source with argument of type `Ptr<const Packet>`, that is, a pointer to a network packet. The second is a trace source with arguments of type `uint32_t`, which indicate the number of bytes in the packet received from the input. The trace sources emit values when either the probed trace source emits a new value, or when `SetValue ()` is called.

Finally, the currently implemented probe classes offer a mechanism to instrument model code directly, in a manner similar to `ns2measure`'s `Stat::Put ()` capability. Each probe, when instantiated, is added to a branch of the configuration namespace. The static method `SetValueByPath (path, value)` takes as argument the path to a specific probe object that is to receive the value, which is provided as the second argument. The effect of `SetValueByPath ()` is analogous to that of the the C programming language's `printf ()` function. Whereas `printf ()` writes to the process' standard output file, `SetValueByPath ()` injects the received value into the probe object named in the first argument, which in its turn pushes out the value through its output trace source.

4.3 Collector

The `Collector` class encapsulates objects that process data received from one or more sources, some of which may be instances of `Probe` or `Collector`. The name of this class stems from this functionality of pulling samples from various objects for manipulation. (Note that this makes it analogous to OSIF's *processors* instead of what it calls *collectors*.)

A `Collector` accepts data from one or more data sources, performs some computation on that data, and reports outputs to some DCF object downstream via its own trace sources. A `Collector` may be connected to a `Probe`. Whenever the trace source in the `Probe` produces a new value, the `Collector` downstream from it consumes the value, processes it, and eventually sends it on to whatever other DCF object follows from it. Objects upstream connect to a `Collector` by establishing the `Collector`'s trace sink method as a callback.

In what regards the timing for the generation of output by a `Collector`, there are two possibilities. If the `Collector` is *periodic*, it produces outputs at regular time intervals defined in its configuration. Otherwise, if the `Collector` is *asynchronous*, it generates output only after having received the number of samples indicated in its configured *batch size*. In this latter case, no output is emitted until a complete batch of samples has been processed.

The completed and ongoing work on collector classes includes the following derivations from the `Collector` base class:

- `BasicStatsCollector` encapsulates the computation of various statistics on a stream of observations. The class defines three trace sources for sample count (the number of observations received), sample sum (the summation of all observations), and sample mean (the mean of all observations). Objects downstream from this collector connect to one of these trace sources by installing their trace sink method as a callback. When that method is invoked, the object can query this collector for the most recent updates to statistics like min, max, standard deviation, and squared sum of observations. An experimental version of this collector provides also autocorrelation and autocovariance for a configurable lag value.
- `BivariateStatsCollector` takes in two streams of observations a and b and applies to them an extended version of Welford's algorithm to compute bivariate statistics (Leemis and Park 2006). It defines two trace sources for output: one with \bar{a} and the other with \bar{b} . Additional statistics such as the sample variance in each of the input streams, the sample covariance, and the sample correlation coefficient can be queried by downstream objects as in the case above.

- `MovingAveragesCollector` takes in a stream of observations as input and provides as output a corresponding stream denoised by the application of a moving average with a configurable window size.
- `SteadyStateCollector` is a base class for the derivation of mechanisms for a combination of steady-state detection and data deletion. It takes in a stream of observations as input and provides two trace sources as output: one of them unfiltered, that is, identical to the input and the other filtered to contain only the observations collected after the identified *warmup* or *transient* period for the observations.
- `MeansCrossingSteadyStateCollector` is a derived class that implements heuristic *R5* for steady-state detection discussed in (Pawlikowski 1990), which is stated as “*The initial transient period is over after n_0 observations if the time series x_1, x_2, \dots, x_{n_0} crosses the mean $\bar{X}(n_0)$ k times.*”
- `Mser5SteadyStateCollector` is a derived class that implements the MSER-5 algorithm for steady-state detection discussed in (White and Robinson 2010).

4.4 Adapter

As we worked in the evaluation of the DCF’s design, we discovered circumstances in which it was necessary to manipulate sample streams so that the data type or *signature* of a trace source output would match that of an input trace sink. The example of note emerged when we attempted to feed the output of a probe with integer signature to an observer that expected input in the form of tuples $\{t, v\}$, where t is a time value and v is a double value. Since all probes currently implemented export trace sources with unidimensional sample values, we realized that users may commonly need to take sample streams from multiple trace sources to build a more complex stream.

The solution for this type of situation is well known as the *adapter* pattern described by Gamma et al. (1995). The class `TimeSeriesAdapter`, which we provide in the DCF, illustrates how to apply this concept in the building of time series with tuples $\{t, v\}$. This class receives each sample v_i from a unidimensional trace sink and invokes a simulator method to get the current clock value t_i . From these two it generates a two-dimensional sample that is exported via its output trace source.

4.5 Aggregator

The role of the `Aggregator` class is to consolidate various sources of data into a coherent whole that can be marshaled into the output format specified by the user. Strictly speaking, an `Aggregator` does not perform any additional filtering or processing of data. However, its operations on data may include annotation, organization, and conversion across different formats. As such, it can be understood as the endpoint for the output that is generated in the simulation.

The inputs for an `Aggregator` are trace sources exported by `Probe` objects and by `Collector` objects. An `Aggregator` provides a trace sink method that is invoked by callback by preceding nodes in the dataflow graph. It may be connected directly to the raw output of a probe and/or connected to receive processed output from a collector. The outputs of an `Aggregator` may be a single or possibly multiple artifacts with types ranging from plot (in some graphic format), to plaintext file, to spreadsheet file, or to insertion queries for a database. Currently implemented aggregator classes are:

- `FileAggregator` stores the data it receives from upstream into a text file. It supports formatted output in the style of C’s `sprintf` function, space, comma, or tab separated files for spreadsheet manipulation as output. It is constrained to receive as input tuples with number of elements in the range 1 to 10 (all of type `double`). The output it generates matches the structure of the input tuples it receives.
- `GnuplotAggregator` receives data from upstream and creates artifacts that are used by the `gnuplot` application to generate static graphs. The input it accepts may be a pairs of point coordinates (x, y) or a 3-tuple (x, y, δ) , when creating plots with symmetric error bars. This

aggregator offers the user a variety of plot configuration options that can be set using methods in this class. Once the simulation terminates, the user finds a number of files left behind as artifacts. Files with names ending in “.plt” are `gnuplot` scripts with all the configuration options specified in the user’s simulation script. They can be modified after the simulation run and before plots are generated, so in the case the user regrets any of the plot configuration options, the simulation wouldn’t have to be executed again. The data sets for the plots are also recorded into their own files. Once the user is ready to build the plots with data created in the simulation, all that needs to be done is the invocation of the shell scripts with names ending in “.sh”. These scripts invoke `gnuplot` to create plots in the graphics format specified by the user’s choice of “terminal” (e.g., `png`, `jpg`, `pdf`, etc.)

- `SafeAggregator` takes the data it receives from upstream and pushes it out onto a Unix *pipe* that connects to SAFE’s *Simulation Client* (SC) process. The ultimate destination of the data provided to this aggregator is SAFE’s database, which holds all the information needed for the reproduction of the simulation as well as its results. The SC receives this output data from this aggregator and relays it to *SAFE Experiment Execution Manager* via a TCP socket, which finally issues a query to record the data in the appropriate database table. At the time of this writing, this class is under development, as the interaction between the *ns-3* simulation and the framework is being refactored to accommodate for changes in the way the database is managed and integrated with user interfaces.

5 A COMPLETE EXAMPLE

In this section, we walk the reader through a complete example of usage of the DCF. We use this opportunity to illustrate the two options of application programming interface (API) that are available to those who write *ns-3* simulation scripts.

5.1 The High-Level DCF API

Figure 1 shows a fragment of code that a user would write into a *ns-3* simulation script for the simulation of two network nodes communicating via TCP/IP over a point-to-point link. This fragment uses a high-level of abstraction API to the DCF implemented according to the *ns-3* tradition a *helper* class to give access to a module’s functionality. A helper class provides APIs that minimize the verbosity and the repetitiveness that can ensue in working with the module’s fine grained, low-level APIs. As described in *ns-3* (2013a) documentation, there is no effort to offer generic helper APIs derived from a common base class. Instead, disjoint helper classes may exist for various modules. Each specific helper class may serve as a container for a group of related objects with methods that automate repetitive actions.

This code fragment illustrates what the user would write in the simulation script (not in model code!) to instantiate probe, collector, and aggregator, and to interconnect them to produce a plot. This example uses the helper’s `ConfigurePlot ()` method to configure a `GnuplotAggregator`, which creates a PDF graphic of a plot showing how the number of packets transmitted by node 0 evolves with time. If the user wants to plot also the evolution of mean packet length (in bytes) over time and the sum of packet length over time, this same fragment is instantiated two more times, with the appropriate changes in arguments to the methods invoked. This helper also includes provisions that allow a “wildcard” to replace the integer number used for node identifier, which causes the output of various probes to be represented as different data series in one same plot. As the full details of this API are beyond the scope of this paper, we refer the interested reader to the *ns-3* documentation.

The ease of use of the high-level API is noteworthy in the example, but also is the fact that the code for data collection and processing exists outside the code of the simulation models. The helper API hides a considerable amount of tedious, repetitive work behind its method calls, but the implementation is not optimal. For each `GnuplotHelper`, a separate probe and collector are instantiated, which is certainly wasteful in comparison to what can be done with the low-level API discussed next.


```

1 // Create the gnuplot helper.
2 GnuplotHelper plotHelper1;
3
4 // Add a probe to the gnuplot helper.
5 plotHelper1.AddProbe ("ns3::Ipv4PacketProbe",
6                      "Node0PacketTxProbe",
7                      "/NodeList/0/$ns3::Ipv4L3Protocol/Tx");
8
9 // Add a collector to the gnuplot helper.
10 plotHelper1.AddCollector ("ns3::BasicStatsCollector",
11                          "Node0PacketTxCollector",
12                          "Node0PacketTxProbe",
13                          "OutputBytes");
14
15 // Configure the plot.
16 plotHelper1.ConfigurePlot ("ipv4-packet-plot-example-packet-count",
17                           "Packet_Count_vs_._Time",
18                           "Time_(Seconds)",
19                           "Packet_Count",
20                           "pdf");

```

Figure 1: Usage example of the high-level helper API for the DCF.

5.2 The Low-Level DCF API

A programmer can use the DCF’s classes directly in their *ns-3* simulation scripts, without going through a helper. The lower-level API gives the user more control over the memory footprint and the execution performance of the DCF. At this level, the user must explicitly create all the instances of DCF objects they need and interconnect them using a bare-bones basic “plumbing” infrastructure. It leads to the repetition of boiler-plate code, but allows the programmer to express the needs of the simulation script directly, with minimal memory and run-time overhead. Figure 2 contains code fragments to achieve the same results obtained with the code in Figure 1. In this code, ellipses are used in place of nearly identical repetitions of the set of statements that precedes them. Figure 3 shows the dataflow graph created in this example and illustrates the plots generated by each `GnuplotAggregator`.

While the low-level API meets the needs of more experienced users of *ns-3*, it may not appeal to those who either don’t have the coding skills or don’t want to incur the extra expense of effort and time to customize their networks of DCF objects.

```

1 // Create the packet probe
2 Ptr<Ipv4PacketProbe> packetProbe = CreateObject<Ipv4PacketProbe>();
3 packetProbe->Enable();
4
5 // Create the collector
6 Ptr<BasicStatsCollector> collector = CreateObject<BasicStatsCollector>();
7 collector->SetPeriodic (Seconds (0.5));
8 collector->Enable();
9
10 // Create the gnuplot aggregator 1
11 Ptr<GnuplotAggregator> gnuplotAgg1 =
12 CreateObject<GnuplotAggregator> ("IPv4.PacketCountPlot");
13 gnuplotAgg1->Set2dDatasetDefaultStyle (Gnuplot2dDataset::LINES);
14 gnuplotAgg1->SetTitle ("Packet_Count_vs_._Time");
15 gnuplotAgg1->SetLegend ("Packet_Count", "Time_(Seconds)");
16 gnuplotAgg1->Add2dDataset ("dataset", "Packet_count");
17 gnuplotAgg1->SetTerminal ("pdf");
18 gnuplotAgg1->Enable();
19 ...
20 // Hook up trace source with probe
21 packetProbe->ConnectByPath ("/NodeList/0/$ns3::Ipv4L3Protocol/Tx");
22
23 // Hook up packet probe with collector
24 packetProbe->TraceConnectWithoutContext ("OutputBytes", MakeCallback (&BasicStatsCollector::TraceSinkUInteger32, collector));
25
26 // Hook up collector with gnuplotAgg1
27 collector->TraceConnect ("SampleCount", "dataset", MakeCallback (&GnuplotAggregator::Write2d, gnuplotAgg1));
28 ...

```

Figure 2: Usage example of the low-level API for the DCF.

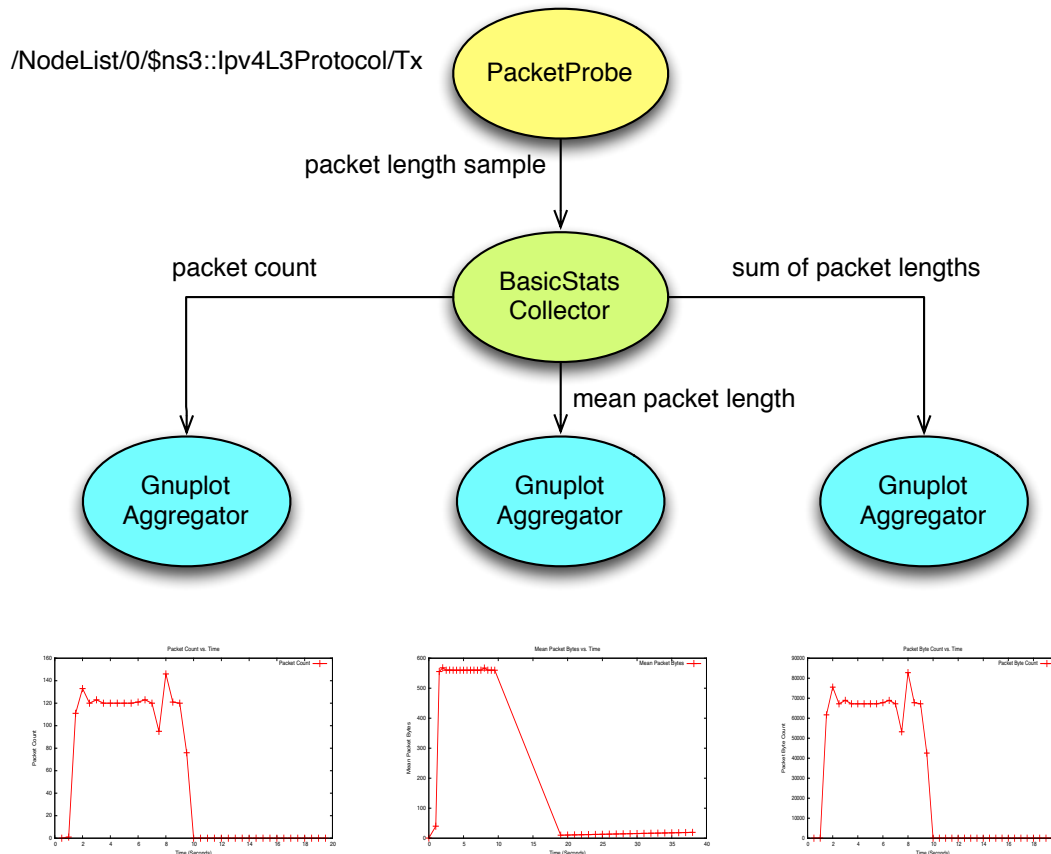


Figure 3: Usage of DCF objects to built three different plots in TCP simulation.

6 CONCLUSION AND FUTURE WORK

This paper presents a flexible and extensible data collection framework for the *ns-3* network simulator, which is expected to be publicly available in the *stats* module of release *ns-3.18*. We have created the basic infrastructure for a module that we expect will serve the community well. The DCF will make it significantly easier for users to write simulation scripts that are able to extract performance attributes of interest from *ns-3* models and to process samples of these data as the simulation executes. This framework doesn't completely do away with the need for a modicum of instrumentation code to be included inside models, however. We expect that model authors will want to add new trace sources to their code, so that users will be able to easily hook them up to DCF structures, when needed.

Going forward, we will want to quantify the performance cost that this data collection and processing will have on the run time execution of the simulation. As discussed by Schützel et al. (2012), since this cost tends to be significant, it is important to understand where bottlenecks lie and work toward their improvement.

One additional enhancement ahead of us, which was not indicated in body of the paper, regards creating the means for objects at the end of the directed dataflow graph to be able to have complete information on the provenance of data. In order to promote a better, long-term understanding of results of a simulation, it will be interesting to instrument *Aggregators* to know the complete path through the *ns-3* namespace that was traversed by data. Having these paths encoded in context strings such as those used to connect model trace sources to probes will allow users to see immediately how data was generated and processed without extensive analysis of the *ns-3* script source code.

ACKNOWLEDGMENTS

This project is supported by the National Science Foundation (NSF) under Grant Nos. CNS-0958139, CNS-0958142, and CNS-0958015. Any opinions, findings, and conclusions or recommendations expressed in this material are the authors' alone and do not necessarily reflect the views of the NSF.

There have been many contributions to the conceptual development and to the implementation of the *ns-3* DCF. We are grateful to Pavel Boyko, Kirill Andreev, and Mathieu Lacage, who contributed with productive discussions and proof-of-concept prototypes that guided the initial development of the project.

Finally, we acknowledge the funding source that supported co-author Vinícius D. Felizardo (in 2013) and collaborator Tiago G. Rodrigues (in 2012), both of whom developed infrastructure for the DCF project. These two undergraduates from Brazil spent a year as exchange students at Bucknell University, which enabled their full-time participation in research and development during summer months. They were supported by the program “Science Without Borders,” a cooperation of the Brazilian Ministry of Science, Technology, and Innovation (MCTI), Ministry of Education (MEC), and funding agencies CAPES and CNPq.

REFERENCES

- Ciconetti, C., E. Mingozzi, and G. Stea. 2006, October. “An Integrated Framework for Enabling Effective Data Collection and Statistical Analysis with *ns-2*”. In *Proceedings of the 2006 workshop on ns-2: the IP network simulator (WNS2 '06)*. Pisa, Italy.
- ns-3* 2013a, May. *The ns-3 Manual (ns-3.17)*. <http://www.nsnam.org/docs/manual/html/index.html> [Accessed May 20, 2013].
- ns-3* 2013b, May. *The ns-3 Tutorial (ns-3.17)*. <http://www.nsnam.org/docs/release/3.17/tutorial/html/index.html> [Accessed May 20, 2013].
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns*. Addison-Wesley.
- Helms, T., J. Himmelspach, C. Maus, O. Röwer, J. Schützel, and A. M. Uhrmacher. 2012, December. “Toward a Language for the Flexible Observation of Simulations”. In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher: IEEE, Piscataway, NJ.
- Kurkowski, S., T. Camp, and M. Colagrosso. 2005. “MANET Simulation Studies: The Incredibles”. *ACM SIGMOBILE Mobile Computing and Communications Review* 9 (4): 50–61.
- Leemis, L. M., and S. K. Park. 2006. *Discrete-Event Simulation: A First Course*, Chapter 4, 172–184. Pearson Education, Inc.
- Pawlikowski, K. 1990, June. “Steady-state simulation of queueing processes: survey of problems and solutions”. *Computing Surveys* 22 (2): 123–170.
- Perrone, L. F., C. S. Main, and B. C. Ward. 2012, December. “SAFE: Simulation Automation Framework for Experiments”. In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher: IEEE, Piscataway, NJ.
- Ribault, J., O. Dalle, D. Conan, and S. Leriche. 2010. “OSIF: A Framework to Instrument, Validate, and Analyze Simulations”. In *Proceedings of the 3rd International Conference on Simulation Tools and Techniques (SIMUTools 2010)*: ICST, Brussels, Belgium.
- Riley, G. F., and T. R. Henderson. 2010. *Modeling and Tools for Network Simulation*, Chapter “The *ns-3* Network Simulator”, 15–34. Springer.
- Schützel, J., J. Himmelspach, H. Meyer, A. Heuer, and A. M. Uhrmacher. 2012, December. “Streaming Data Management for the Online Processing of Simulation Data”. In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher: IEEE, Piscataway, NJ.

- Uhrmacher, A. 2012, December. “Seven Pitfalls in Modeling and Simulation Research”. In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher: IEEE, Piscataway, NJ.
- White, K., and S. Robinson. 2010. “The Problem of the Initial Transient (Again) or Why MSER Works”. *Journal of Simulation* 4:268–272.

AUTHOR BIOGRAPHIES

L. FELIPE PERRONE is Associate Professor of Computer Science at Bucknell University. He holds Ph.D. and M.Sc. degrees from the College of William & Mary (USA), as well as an M.Sc. and the degree of Electrical Engineer from the Universidade Federal do Rio de Janeiro (Brazil). He is an Associate Editor of the ACM TOMACS and has served the simulation community in several organizational roles in conferences, including Proceedings Editor of the Winter Simulation Conference 2006, and Program and General Chair of SIMUTools (2009 and 2010, respectively). His interests include modeling and simulation, computer networks, and high performance computing. His email address is perrone@bucknell.edu and his web page is <http://www.eg.bucknell.edu/~perrone>.

THOMAS R. HENDERSON is Affiliate Professor of Electrical Engineering at the University of Washington and Technical Fellow with Boeing Research & Technology. He is a founder and the open source project lead for the *ns-3* project. His email address is tomhend@u.washington.edu.

MITCHELL J. WATROUS is Software Engineer at the University of Washington. He holds Ph.D. and M.Sc. degrees in Physics from the University of Washington. He is currently working on the *ns-3* project. His email address is watrous@u.washington.edu.

VINÍCIUS DALY FELIZARDO pursues the degree of Computer Engineer at the Universidade Estadual de Campinas (UNICAMP), in Campinas, Brazil. He spent one year as exchange student at Bucknell University, during which he applied himself to the learning and the practice of computer network simulation. He is a major contributor in the *ns-3* SAFE project. His email address is vd001@bucknell.edu.